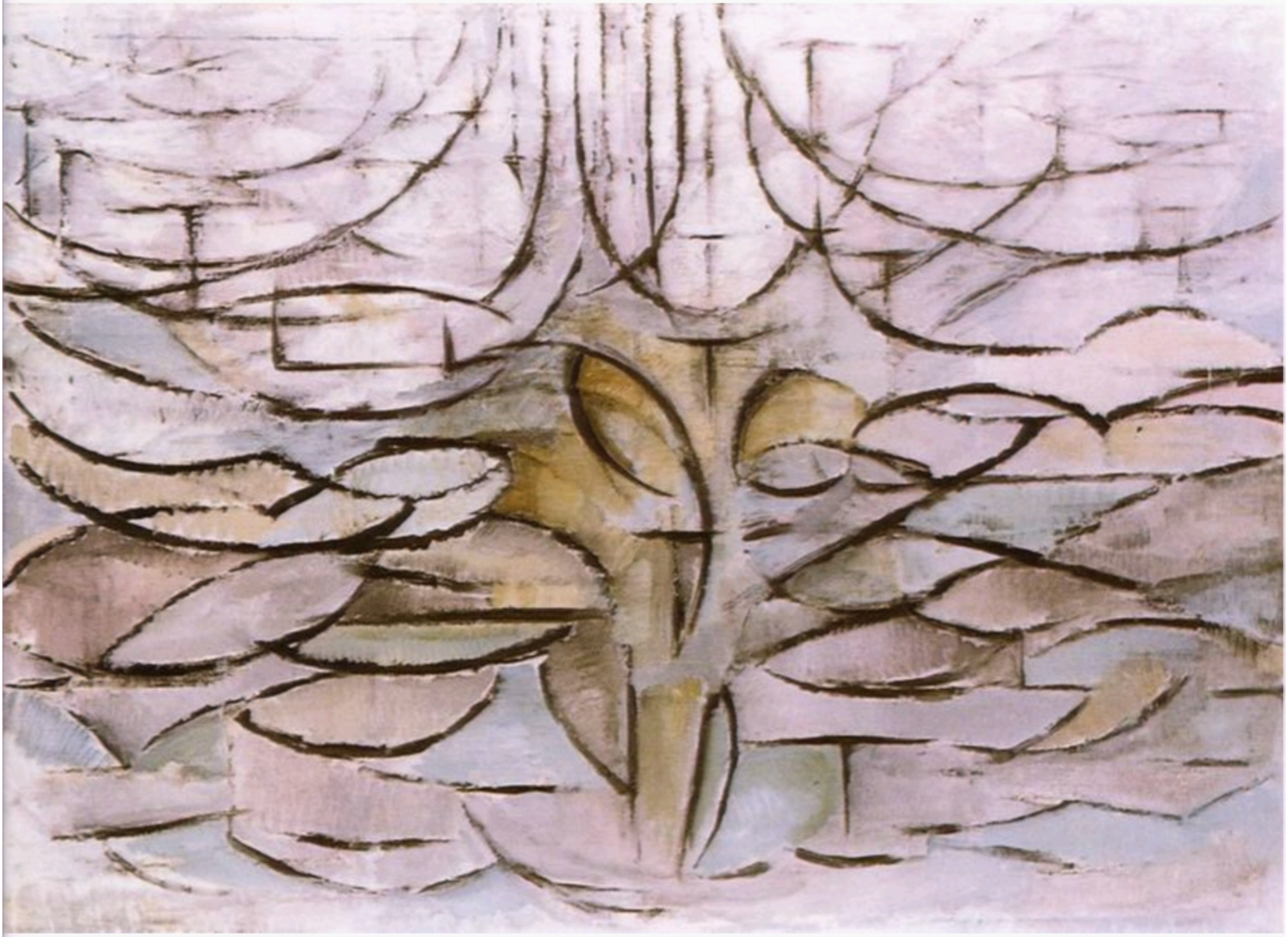
Dimensions of *Duality* in Data Models

Erik Meijer
co-Researcher



Dimensions of *Duality* in Data Models

Erik Meijer
co-Researcher

Categories, objects, and morphisms

Main articles: Category (mathematics) and Morphism

A *category* C consists of the following three mathematical entities:

- A class ob(C), whose elements are called *objects*;
- A class hom(C), whose elements are called morphisms or maps or *arrows*. Each morphism f has a unique *source object* a and *target object* b. We write $f: a \to b$, and we say "f is a morphism from a to b". We write hom(a, b) (or Hom(a, b), or $\hom_C(a, b)$, or Mor(a, b), or $C(a, b)$) to denote the *hom-class* of all morphisms from a to b.
- A binary operation $\circ$, called *composition of morphisms*, such that for any three objects a, b, and c, we have hom(a, b) × hom(b, c) → hom(a, c). The composition of $f: a \to b$ and $g: b \to c$ is written as $g \circ f$ or gf[2], governed by two axioms:
  - Associativity: If $f : a \to b$, $g : b \to c$ and $h : c \to d$ then $h \circ (g \circ f) = (h \circ g) \circ f$, and
  - Identity: For every object x, there exists a morphism $1_x : x \to x$ called the *identity morphism for* x, such that for every morphism $f : a \to b$, we have $1_b \circ f = f = f \circ 1_a$.

From these axioms, it can be proved that there is exactly one identity morphism for every object. Some authors deviate from the definition just given by identifying each object with its identity morphism.

Relations among morphisms (such as $fg = h$) are often depicted using commutative diagrams, with "points" (corners) representing objects and "arrows" representing morphisms.

The definitions of categories and functors provide only the very basics of categorical algebra; additional important topics are listed below. Although there are strong interrelations between all of these topics, the given order can be considered as a guideline for further reading.

- The functor category D^C has as objects the functors from C to D and as morphisms the natural transformations of such functors. The Yoneda lemma is one of the most famous basic results of category theory; it describes representable functors in functor categories.
- Duality: Every statement, theorem, or definition in category theory has a *dual* which is essentially obtained by "reversing all the arrows". If one statement is true in a category C then its dual will be true in the dual category C^{op}. This duality, which is transparent at the level of category theory, is often obscured in applications and can lead to surprising relationships.
- Adjoint functors: A functor can be left (or right) adjoint to another functor that maps in the opposite direction. Such a pair of adjoint functors typically arises from a construction defined by a universal property; this can be seen as a more abstract and powerful view on universal properties.

Initial Algebras
Versus
Final coAlgebras

Bird-Meertens Formalism

From Wikipedia, the free encyclopedia

The **Bird-Meertens Formalism** is a calculus for deriving programs from specifications (in a functional-programming setting), devised by Richard Bird and Lambert Meertens.

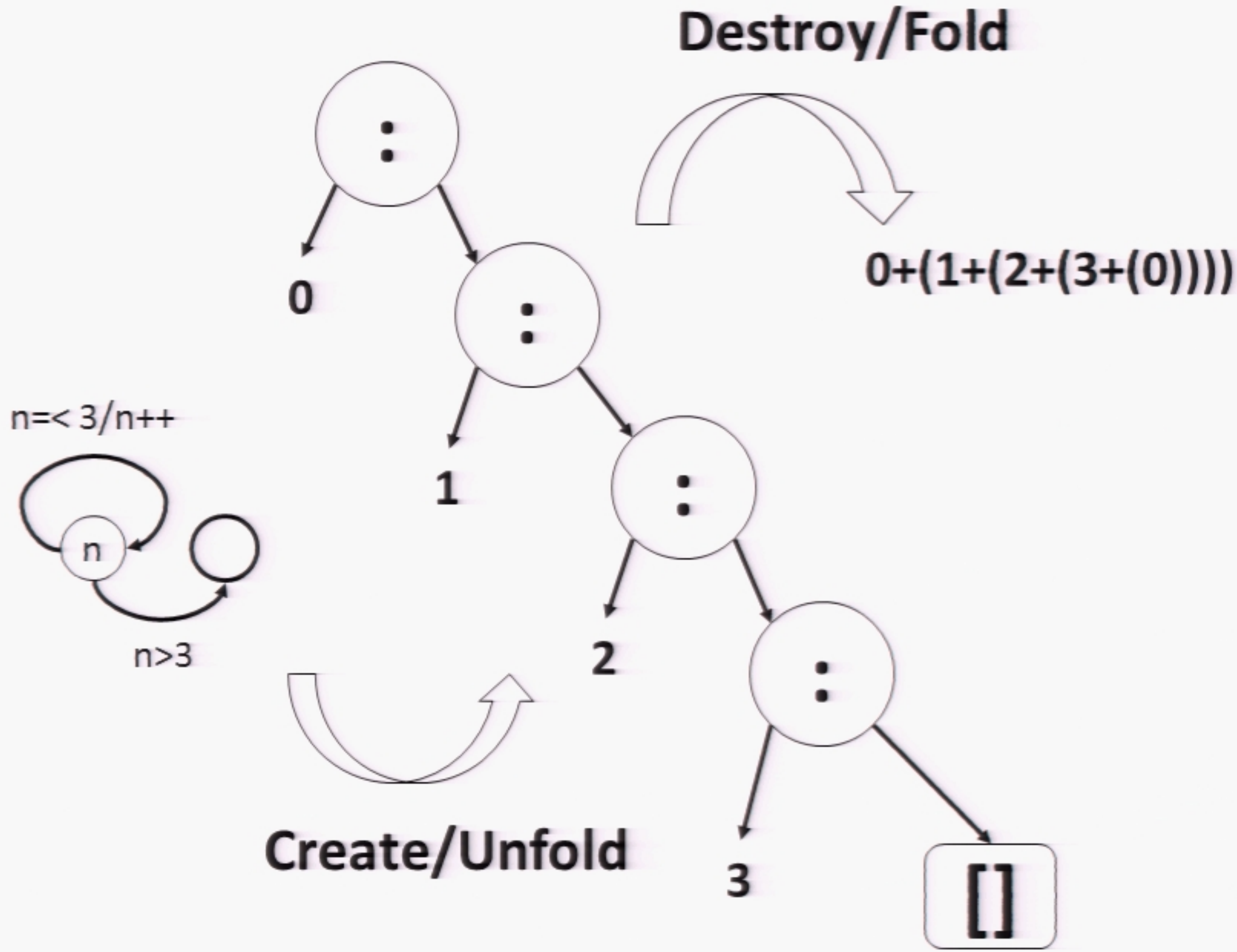
It is sometimes facetiously known as **Squiggol**, because of the "squiggly" symbols it uses. A less-used variant name, but actually the first one suggested, is **SQUIGOL**.

See also [edit]

- Catamorphism
- Anamorphism
- Paramorphism
- Hylomorphism

References [edit]

- Richard Bird; Oege de Moor (1997). *Algebra of Programming, International Series in Computing Science, Vol. 100.* Prentice Hall. ISBN 0-13-507245-X.


Destroy/Fold
0+(1+(2+(3+(0))))
:
0
:
1
:
2
:
3
[]
n=< 3/n++
n
n>3
Create/Unfold

Bird’s First Homomorphism Lemma

1987

A function

$h :: [A] \rightarrow B$

is a homomorphism wrt to $\cup$ iff

$h = (\oplus/) \bullet (f*)$ -- “/” is reduce, “*” is map

for some

$f :: A \rightarrow B$

and

$\oplus :: BxB \rightarrow B$

Initial algebra

From Wikipedia, the free encyclopedia

In mathematics, an **initial algebra** is an initial object in the category of *F*-algebras for a given endofunctor *F*. The initiality provides a general framework for induction and recursion.

For instance, consider the endofunctor 1+(-) on the category of sets, where 1 is the one-point set, the terminal object in the category. An algebra for this endofunctor is a set X (called the *carrier* of the algebra) together with a point $x \in X$ and a function $X \to X$. The set of natural numbers is the carrier of the initial such algebra: the point is zero and the function is the successor map.

For a second example, consider the endofunctor $1+\mathbf{N}\times(-)$ on the category of sets, where **N** is the set of natural numbers. An algebra for this endofunctor is a set X together with a point $x \in X$ and a function $\mathbf{N}\times X \to X$. The set of finite lists of natural numbers is the initial such algebra. The point is the empty list, and the function is cons, taking a number and a finite list, and returning a new finite list with the number at the head.

Contents [hide]

1 Final coalgebra
2 Theorems
3 Example
4 Use in Computer Science
5 See also
6 Notes
7 External links

Final coalgebra

[edit]

Dually, a **final coalgebra** is a terminal object in the category of *F*-coalgebras. The finality provides a general framework for coinduction and corecursion.

For example, using the same functor 1+(-) as before, a coalgebra is a set X together with a truth-valued test function $p : X \to 2$ and a partial function $f : X \to X$ whose domain is formed by those $x \in X$ for which $p(x) = 0$. The set $\mathbf{N} \cup \{\omega\}$ consisting of the natural numbers extended with a new element ω is the carrier of the final coalgebra in the category, where p is the test for zero: $p(0) = 1$, $p(n+1) = p(\omega) = 0$; and f is the predecessor function (the inverse of the successor function) on the positive naturals, but acts like the identity on the new element ω: $f(n+1) = n$, $f(\omega) = \omega$.

For a second example, consider the same functor $1+\mathbf{N}\times(-)$ as before. In this case the carrier of the final coalgebra consists of all lists *of natural numbers*, finite as well as infinite. The operations are a test function testing whether a list is empty, and a deconstruction function defined on nonempty lists returning a pair consisting of the head and the tail of the input list.

Bird's First Homomorphism Lemma

1987

A function

$h :: [A] \rightarrow B$

is a homomorphism wrt to $\cup$ iff

$h = (\oplus/) \bullet (f*)$ -- "/" is reduce, "*" is map

for some

$f :: A \rightarrow B$

and

$\oplus :: BxB \rightarrow B$

Initial algebra

From Wikipedia, the free encyclopedia

In mathematics, an **initial algebra** is an initial object in the category of *F*-algebras for a given endofunctor *F*. The initiality provides a general framework for induction and recursion.

For instance, consider the endofunctor 1+(-) on the category of sets, where 1 is the one-point set, the terminal object in the category. An algebra for this endofunctor is a set X (called the *carrier* of the algebra) together with a point $x \in X$ and a function $X \to X$. The set of natural numbers is the carrier of the initial such algebra: the point is zero and the function is the successor map.

For a second example, consider the endofunctor $1+\mathbf{N}\times(-)$ on the category of sets, where **N** is the set of natural numbers. An algebra for this endofunctor is a set X together with a point $x \in X$ and a function $\mathbf{N}\times X \to X$. The set of finite lists of natural numbers is the initial such algebra. The point is the empty list, and the function is cons, taking a number and a finite list, and returning a new finite list with the number at the head.

Contents [hide]

1 Final coalgebra
2 Theorems
3 Example
4 Use in Computer Science
5 See also
6 Notes
7 External links

Final coalgebra [edit]

Dually, a **final coalgebra** is a terminal object in the category of *F*-coalgebras. The finality provides a general framework for coinduction and corecursion.

For example, using the same functor 1+(-) as before, a coalgebra is a set X together with a truth-valued test function $p : X \to 2$ and a partial function $f : X \to X$ whose domain is formed by those $x \in X$ for which $p(x) = 0$. The set $\mathbf{N} \cup \{\omega\}$ consisting of the natural numbers extended with a new element ω is the carrier of the final coalgebra in the category, where p is the test for zero: $p(0) = 1$, $p(n+1) = p(\omega) = 0$; and f is the predecessor function (the inverse of the successor function) on the positive naturals, but acts like the identity on the new element ω: $f(n+1) = n$, $f(\omega) = \omega$.

For a second example, consider the same functor $1+\mathbf{N}\times(-)$ as before. In this case the carrier of the final coalgebra consists of all lists of natural numbers, finite as well as infinite. The operations are a test function testing whether a list is empty, and a deconstruction function defined on nonempty lists returning a pair consisting of the head and the tail of the input list.

Ana = upwards
Cata = downwards

$$
\begin{tikzcd}
\mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[d, "F\llbracket\psi\rrbracket"] & \mathbb{N} \arrow[l, "\psi"'] \arrow[d, "\llbracket\psi\rrbracket"] \\
\mathbb{1} + \mathbb{N} \times \mathrm{List}\langle\mathbb{N}\rangle \arrow[d, "F(\!|\varphi|\!)"] \arrow[r, leftrightarrow] & \mathrm{List}\langle\mathbb{N}\rangle \arrow[d, "(\!|\varphi|\!)"] \\
\mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[r, "\varphi"] & \mathbb{N}
\end{tikzcd}
$$

F-coAlgebra

anamorphism

Final F-Algebra

Initial F-Algebra

catamorphism

F-Algebra

$$
\begin{tikzcd}
\text{Initial Algebra} \arrow[d, shift right=4ex, "\text{Introduce } \perp"'] \\
\text{Final coAlgebra} \arrow[u, shift right=4ex, "\text{Remove } \perp"'] \arrow[u, phantom, "\cong"]
\end{tikzcd}
$$

Fusion Laws

$$\frac{f \circ \varphi = \psi \circ Ff}{f \circ (\!|\varphi|\!) = (\!|\psi|\!)}$$

$$\frac{Ff \circ \psi = \varphi \circ f}{[\![\psi]\!] = [\![\varphi]\!] \circ f}$$

Denotational Semantics

$$
\begin{aligned}
\texttt{Expr} \; ::= \; & \texttt{Expr} \oplus \texttt{Expr} \\
\mid \; & \mathbb{N}
\end{aligned}
$$

$$\llparenthesis (a,b) \Rightarrow a+b,\ n \Rightarrow n \rrparenthesis \in \texttt{Expr} \longrightarrow \mathbb{N}$$

$$
\begin{aligned}
\mathcal{E}[\![a \oplus b]\!] &= \mathcal{E}[\![a]\!] + \mathcal{E}[\![b]\!] \\
\mathcal{E}[\![n]\!] &= n
\end{aligned}
$$

Fusion Laws

$$\frac{f \circ \varphi = \psi \circ F f}{f \circ (\!|\varphi|\!) = (\!|\psi|\!)}$$

$$\frac{F f \circ \psi = \varphi \circ f}{[\![\psi]\!] = [\![\varphi]\!] \circ f}$$

Continuation Semantics

$F \in \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
$F\ n\ \sigma = \sigma(n)$

$F\mathcal{E}[\![a \oplus b]\!]\ \sigma$
$=$
$F(\mathcal{E}[\![a]\!]+\mathcal{E}[\![b]\!])\ \sigma$
$=$
$\sigma(\mathcal{E}[\![a]\!]+\mathcal{E}[\![b]\!])$
$=$
$F\mathcal{E}[\![b]\!]\ (y \Rightarrow \sigma(\mathcal{E}[\![a]\!]+y))$
$=$
$F\mathcal{E}[\![a]\!]\ (x \Rightarrow F\mathcal{E}[\![b]\!]\ (y \Rightarrow \sigma(x+y)))$

$F\mathcal{E}[\![n]\!]\ \sigma$
$=$
$F(n)\ \sigma$
$=$
$\sigma(n)$

Operational Semantics

$$\frac{}{n \leadsto n} \qquad \frac{}{n \oplus m \leadsto n+m}$$

$$\frac{a \leadsto c}{a \oplus b \leadsto c \oplus b} \qquad \frac{b \leadsto c}{a \oplus b \leadsto a \oplus c}$$

$$(\underline{1 \oplus 2}) \oplus (3 \oplus 4) \longrightarrow 3 \oplus (\underline{3 \oplus 4}) \longrightarrow \underline{3 \oplus 7} \longrightarrow 10$$

$$(1 \oplus 2) \oplus (\underline{3 \oplus 4}) \longrightarrow (\underline{1 \oplus 2}) \oplus 7 \longrightarrow \underline{3 \oplus 7} \longrightarrow 10$$

Operational Semantics

Transition system defines an anamorphism

```
⟦f,g⟧ ∈ Expr⟶Tree Expr
```

```
Tree a ::= Node a [Tree a]
```

```
⟦b⟶a,b⟶[b]⟧ ∈ b⟶Tree a
⟦f,g⟧ b = Node (f b) (map ⟦f,g⟧ (g b))
```

Operational Semantics

$[\![\text{id, steps}]\!] \in \text{Expr} \longrightarrow \text{Tree Expr}$

$\text{steps} \in \text{Expr} \longrightarrow [\text{Expr}]$

$\text{steps } n = [n]$

$\text{steps } (n \oplus m) = [n+m]$

$$
\begin{aligned}
\text{steps } (a \oplus b) = {} & \text{map } (c \Rightarrow c \oplus b) \text{ (steps } a) \\
& {+}{+} \\
& \text{map } (c \Rightarrow a \oplus c) \text{ (steps } b)
\end{aligned}
$$

Operational Semantics

$$\frac{}{n \rightsquigarrow n} \qquad \frac{}{n \oplus m \rightsquigarrow n+m}$$

$$\frac{a \rightsquigarrow c}{a \oplus b \rightsquigarrow c \oplus b} \qquad \frac{b \rightsquigarrow c}{a \oplus b \rightsquigarrow a \oplus c}$$

$$(\underline{1 \oplus 2}) \oplus (3 \oplus 4) \longrightarrow 3 \oplus (\underline{3 \oplus 4}) \longrightarrow \underline{3 \oplus 7} \longrightarrow 10$$

$$(1 \oplus 2) \oplus (\underline{3 \oplus 4}) \longrightarrow (\underline{1 \oplus 2}) \oplus 7 \longrightarrow \underline{3 \oplus 7} \longrightarrow 10$$

Operational Semantics

$[\![\text{id, steps}]\!] \in \text{Expr} \longrightarrow \text{Tree Expr}$

$$
\begin{aligned}
&\text{steps} \in \text{Expr} \longrightarrow [\text{Expr}] \\
&\text{steps } n = [n] \\
&\text{steps } (n \oplus m) = [n+m] \\
&\text{steps } (a \oplus b) = \text{map } (c \Rightarrow c \oplus b)\ (\text{steps } a) \\
&\qquad\qquad\qquad ++ \\
&\qquad\qquad\qquad \text{map } (c \Rightarrow a \oplus c)\ (\text{steps } b)
\end{aligned}
$$

Operational Semantics

$$\frac{}{n \leadsto n} \qquad \frac{}{n \oplus m \leadsto n+m}$$

$$\frac{a \leadsto c}{a \oplus b \leadsto c \oplus b} \qquad \frac{b \leadsto c}{a \oplus b \leadsto a \oplus c}$$

$$(\underline{1 \oplus 2}) \oplus (3 \oplus 4) \longrightarrow 3 \oplus (\underline{3 \oplus 4}) \longrightarrow \underline{3 \oplus 7} \longrightarrow 10$$

$$(1 \oplus 2) \oplus (\underline{3 \oplus 4}) \longrightarrow (\underline{1 \oplus 2}) \oplus 7 \longrightarrow \underline{3 \oplus 7} \longrightarrow 10$$

Operational Semantics

$[\![\mathtt{id},\ \mathtt{steps}]\!] \in \mathtt{Expr} \longrightarrow \mathtt{Tree\ Expr}$

$\mathtt{steps} \in \mathtt{Expr} \longrightarrow [\mathtt{Expr}]$

$\mathtt{steps}\ n = [n]$

$\mathtt{steps}\ (n \oplus m) = [n+m]$

$\mathtt{steps}\ (a \oplus b) = \mathrm{map}\ (c \Rightarrow c \oplus b)\ (\mathrm{steps}\ a)$

$\qquad +\!+$

$\qquad \mathtt{map}\ (c \Rightarrow a \oplus c)\ (\mathrm{steps}\ b)$

Monads

vs

coMonads

Monads
vs
coMonads

Monads as Kleisli triples

Rather than focusing on a specific T, we want to find the general properties common to all notions of computation, therefore we impose as only requirement that *programs* should form a category. The aim of this section is to convince the reader, with a sequence of informal argumentations, that such a requirement amounts to say that T is part of a Kleisli triple $(T, \eta, \_^*)$ and that the category of programs is the Kleisli category for such a triple.

Definition 1.2 ([**Man76**]) *A* **Kleisli triple** *over a category* $\mathcal{C}$ *is a triple* $(T, \eta, \_^*)$*, where* $T: \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{C})$*,* $\eta_A: A \to TA$ *for* $A \in \mathrm{Obj}(\mathcal{C})$*,* $f^*: TA \to TB$ *for* $f: A \to TB$ *and the following equations hold:*

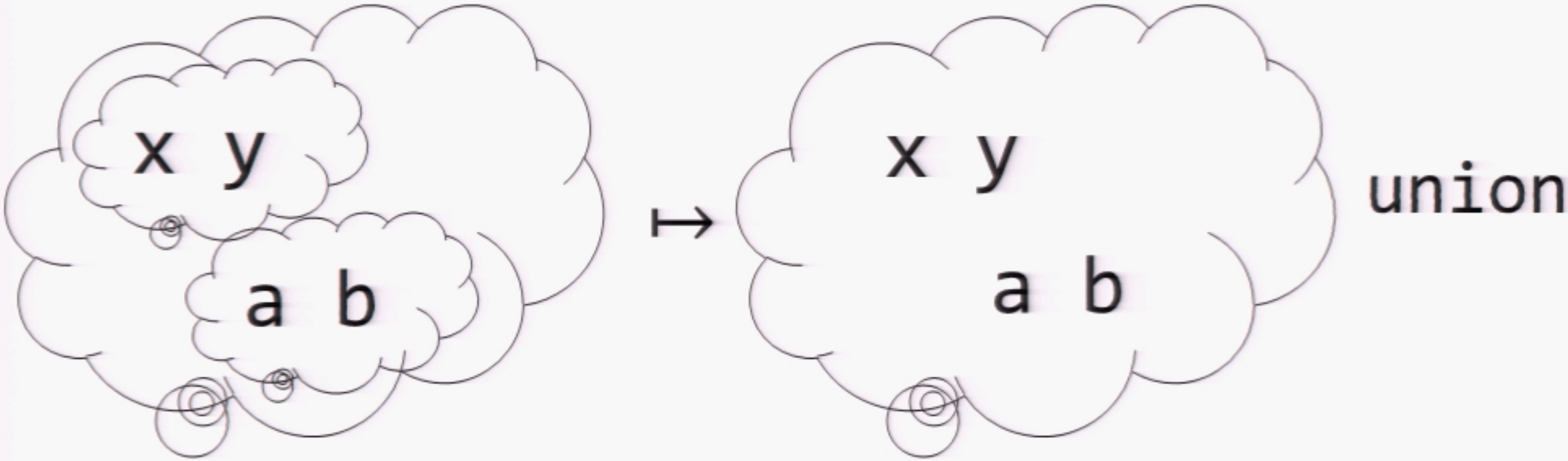
- $\eta_A^* = \mathrm{id}_{TA}$
- $\eta_A; f^* = f$ *for* $f: A \to TB$
- $f^*; g^* = (f; g^*)^*$ *for* $f: A \to TB$ *and* $g: B \to TC$.

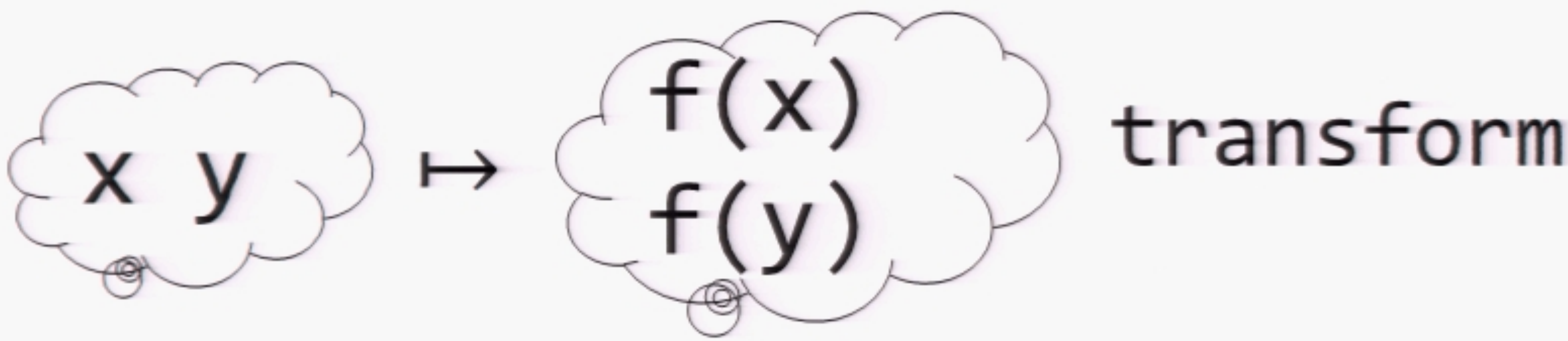
A Kleisli triple satisfies the **mono requirement** *provided* η_A *is mono for* $A \in \mathcal{C}$.

Intuitively η_A is the *inclusion* of values into computations (in several cases η_A is indeed a mono) and f^* is the *extension* of a function f from values to computations to a function from computations to computations, which first evaluates a computation and then applies f to the resulting value. In

What is the most abstract interface for the notion of “collection”?

Works naturally for many





That is a monad

$$\eta \in A \longrightarrow M\langle A\rangle$$
$$\mu \in M\langle M\langle A\rangle\rangle \longrightarrow M\langle A\rangle$$
$$M \in (A \rightarrow B) \longrightarrow (M\langle A\rangle \rightarrow M\langle B\rangle)$$

(plus some obvious coherence conditions)

invocation is a Monad

```
Invoke ∈ (A→IO<B>)×IO<A>⟶IO<B>
return ∈ A⟶IO<A>
```

Effect is implicit

```
Func<A,IO<B>> = a => Bar(a);
var b = Foo.Invoke(ma);
```

LINQ is Monads

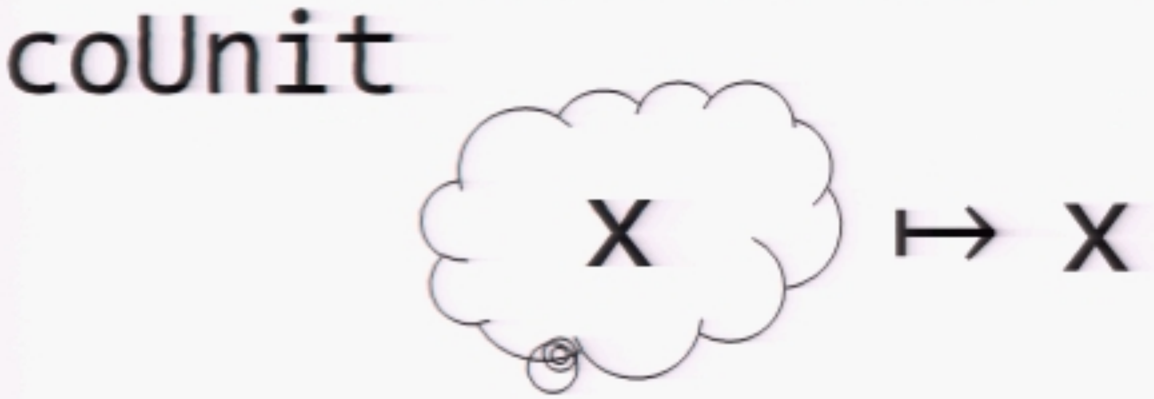
```
from x in xs
where P(x)
let y = F(x)
group G(y) by H(y) into z
select K(z)
```

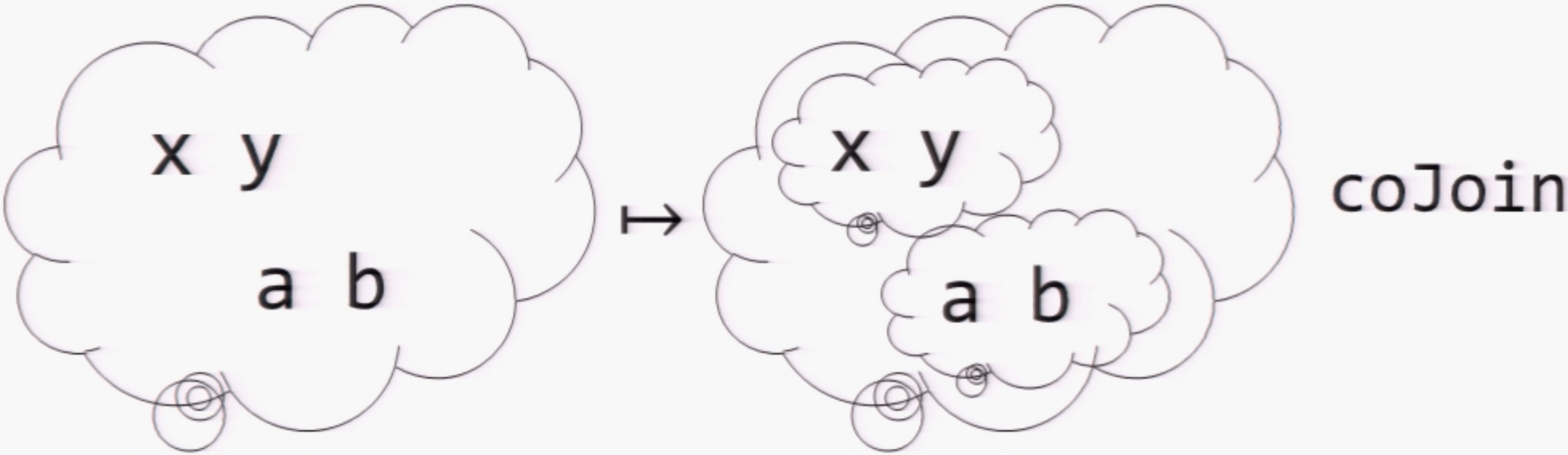
C# 3.0
monad
comprehension

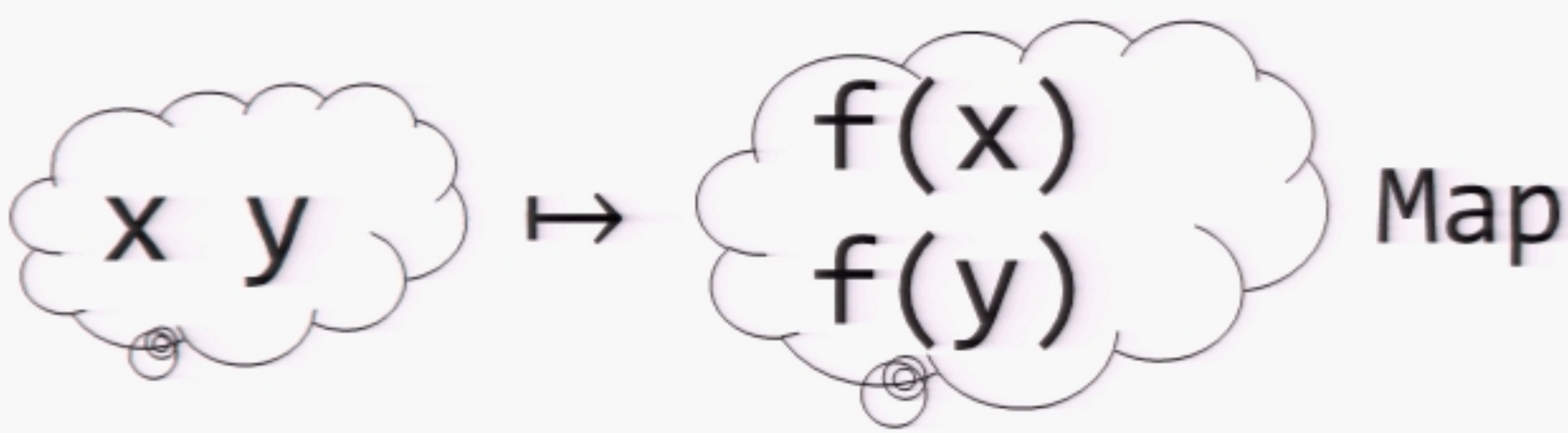
$\mathtt{SelectMany} \in$

$$M\langle A\rangle \times (A \to M\langle B\rangle) \times (A \times B \to C) \longrightarrow M\langle C\rangle$$

Works naturally for one







That is a co-monad

$$\varepsilon \in M\langle A\rangle \longrightarrow A$$

$$\delta \in M\langle A\rangle \longrightarrow M\langle M\langle A\rangle\rangle$$

$$M \in (A \to B) \longrightarrow (M\langle A\rangle \to M\langle B\rangle)$$

(plus some obvious coherence conditions)

TPL Task is a coMonad ☺

```
ContinueWith ∊ Task<A>×(Task<A>→B)⟶Task<B>
Result ∊ Task<A>⟶A
```

```
Task<B> Foo(Task<A> ma)
{
   var a = await(ma);
   return Bar(a);
}
```

C# 5.0
coMonad
comprehension

noSQL

is

coSQL

FK/PK store (SQL)
Versus
Key-Value store (coSQL)

EDM example of SQL data model

The entity type is the fundamental building block for describing the structure of data with the Entity Data Model. […] Each entity must have a unique entity key within an entity set. An entity set is a collection of instances of a specific entity type. Entity sets (and association sets) are logically grouped in an entity container.

REST example of coSQL data model

Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, perhaps, some HTML, XML or JSON that represents some database records expressed, for instance, in Finnish and encoded in UTF-8, depending on the details of the request and the server implementation.

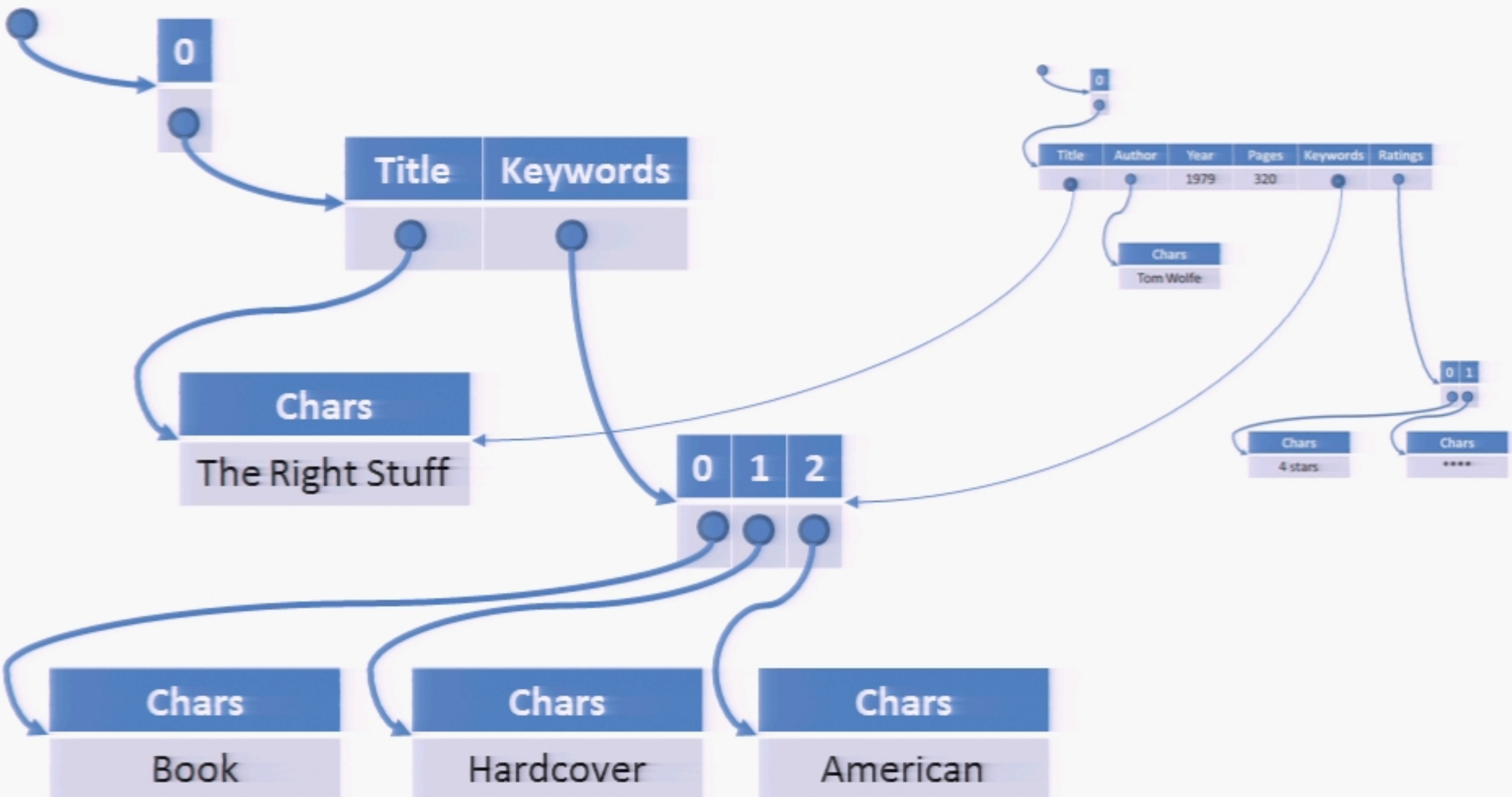
Amazon SimpleDB Sample Query Dataset

```
class Product
{
   string Title;
   string Author;
   int Year;
   int Pages;
   IEnumerable<string> Keywords;
   IEnumerable<string> Ratings;
}

var _1579124585 = new Product
{
  Title = “The Right Stuff”,
  Author = “Tom Wolfe”,
  Year = 1979,
  Pages = 304,
  Keywords = new[]{ “Book”, “Hardcover”, “American” },
  Ratings = new[]{ “****”, “4 stars” },
}

var Products = new[]{ _1579124585 };
```

```
var q = from product in Products
        where product.Ratings.Any(rating => rating == "****")
        select new{ product.Title, product.Keywords };
```



```
table Products
{
  int ID;
  string Title;
  string Author;
  int Year;
  int Pages;
}

table Keywords
{
  int ID;
  string Keyword;
  int ProductID;
}

table Ratings
{
  int ID;
  string Rating;
  int ProductID;
}
```

```
Products.Insert
          ( 1579124585
          , "Tom Wolfe"
          , 1979
          , 304
          );

Keywords.Insert
          ( 4711
          , "Book"
          , 1579124585
          );
Keywords.Insert
          ( 1843
          , "Hardcover"
          , 1579124585
          );
Keywords.Insert
          ( 2012
          , "American"
          , 1579124585
          );
```

```
Ratings.Insert
          ( 787
          , "****"
          , 1579124585
          );
Ratings.Insert
          ( 747
          , "4 stars"
          , 1579124585
          );
```

In SQL rows
are not expressible

ID	Rating	ProductID
787	****	1579124585
747	4 stars	1579124585

ID	Title	Author	Year	Pages
1579124585	The Right Stuff	Tom Wolfe	1979	304

ID	Keyword	ProductID
4711	Book	1579124585
1843	Hardcover	1579124585
2012	American	1579124585

```
var q = from product in Products
        from rating in Ratings
        where product.ID == rating.ProductId
              && rating == "****"
        from keyword in Keywords
        where product.ID == keyword.ProductID
        select new{ product.Title, keyword.Keyword };
```

Title	Keyword
The Right Stuff	Book
The Right Stuff	Hardcover
The Right Stuff	American

```
var q = from product in Products
        join rating in Ratings
        on product.ID equals rating.ProductId
        where rating == "****"
        select product into FourStarProducts
        from fourstarproduct in FourStarProducts
        join keyword in Keywords
        on product.ID equals keyword.ProductID
        select new{ product.Title, keyword.Keyword };
```

ID	Rating	ProductID
787	****	1579124585
747	4 stars	1579124585

ID	Title	Author	Year	Pages
1579124585	The Right Stuff	Tom Wolfe	1979	304

ID	Keyword	ProductID
4711	Book	1579124585
1843	Hardcover	1579124585
2012	American	1579124585

Title	Author	Year	Pages	Keywords	Ratings
The Right Stuff	Tom Wolfe	1979	320		

Chars
American

Chars
Hardcover

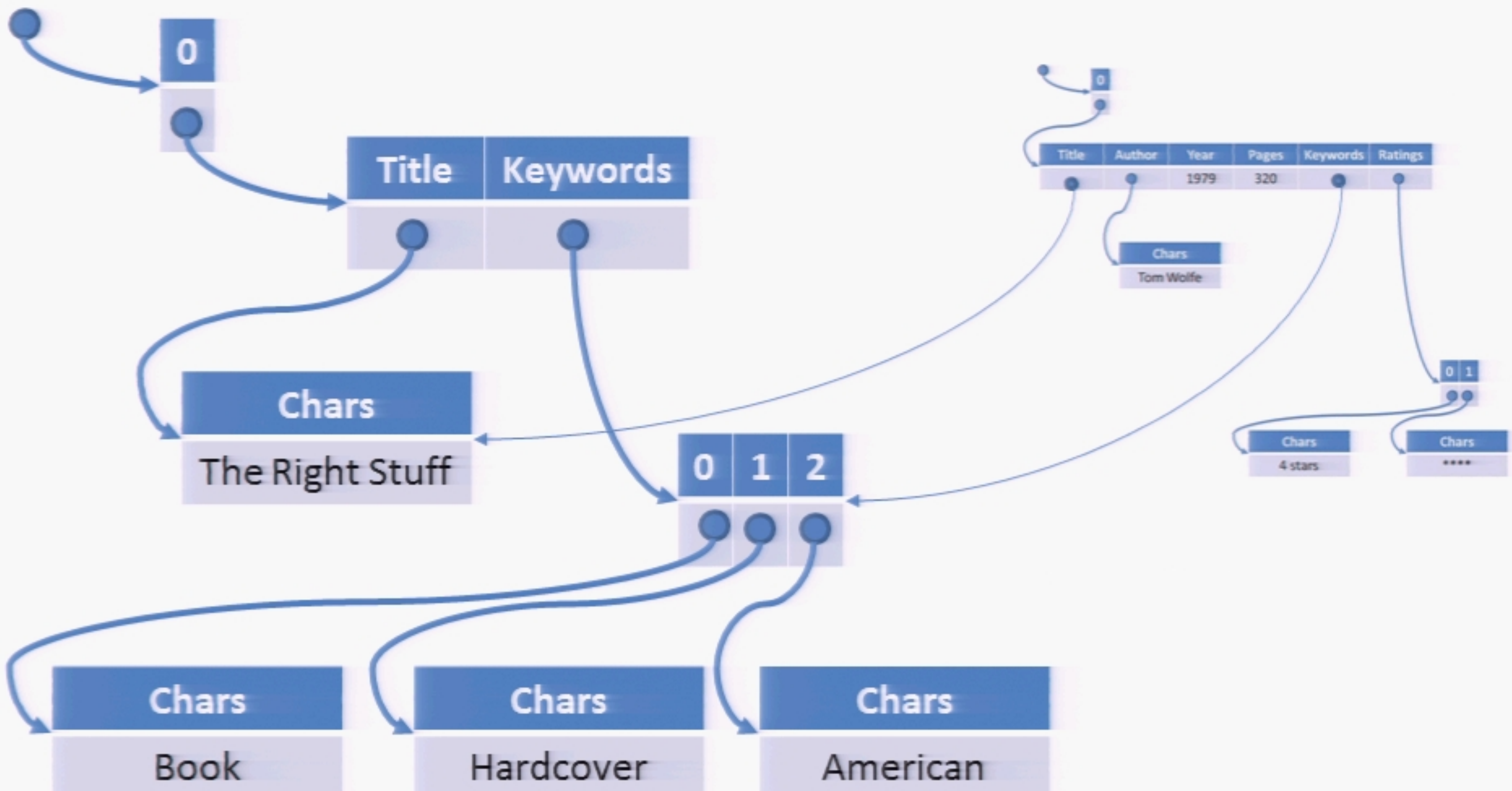
Chars
Book

Chars
****

Chars
4 stars

Arrows Are Reversed

```
var q = from product in Products
        where product.Ratings.Any(rating => rating == "****")
        select new{ product.Title, product.Keywords };
```



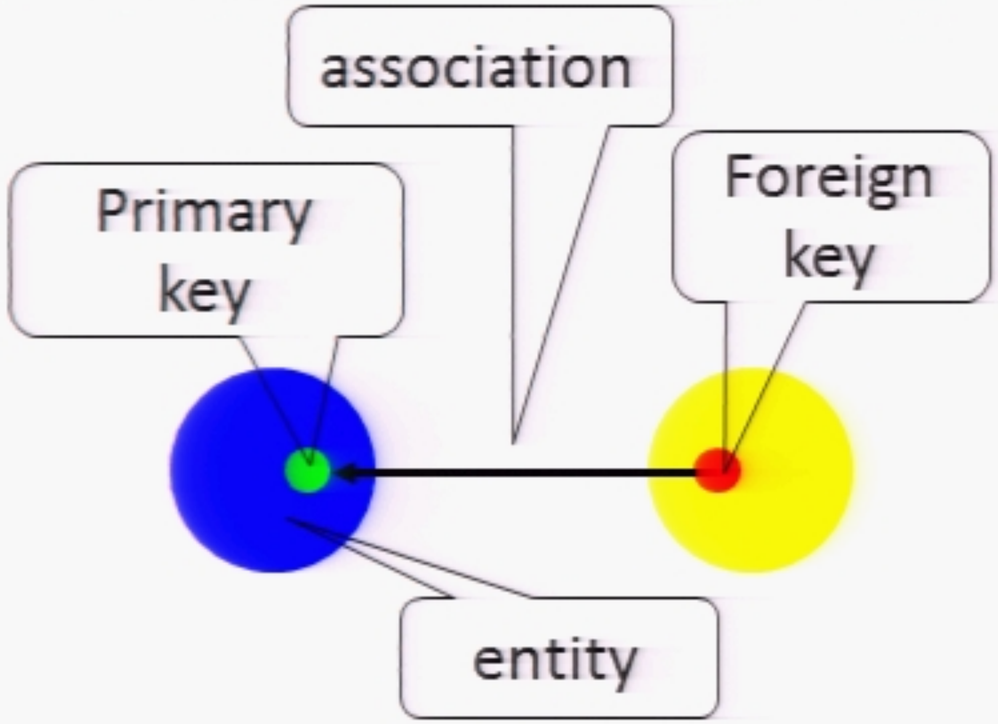
Arrows Are Reversed

Parent points to children

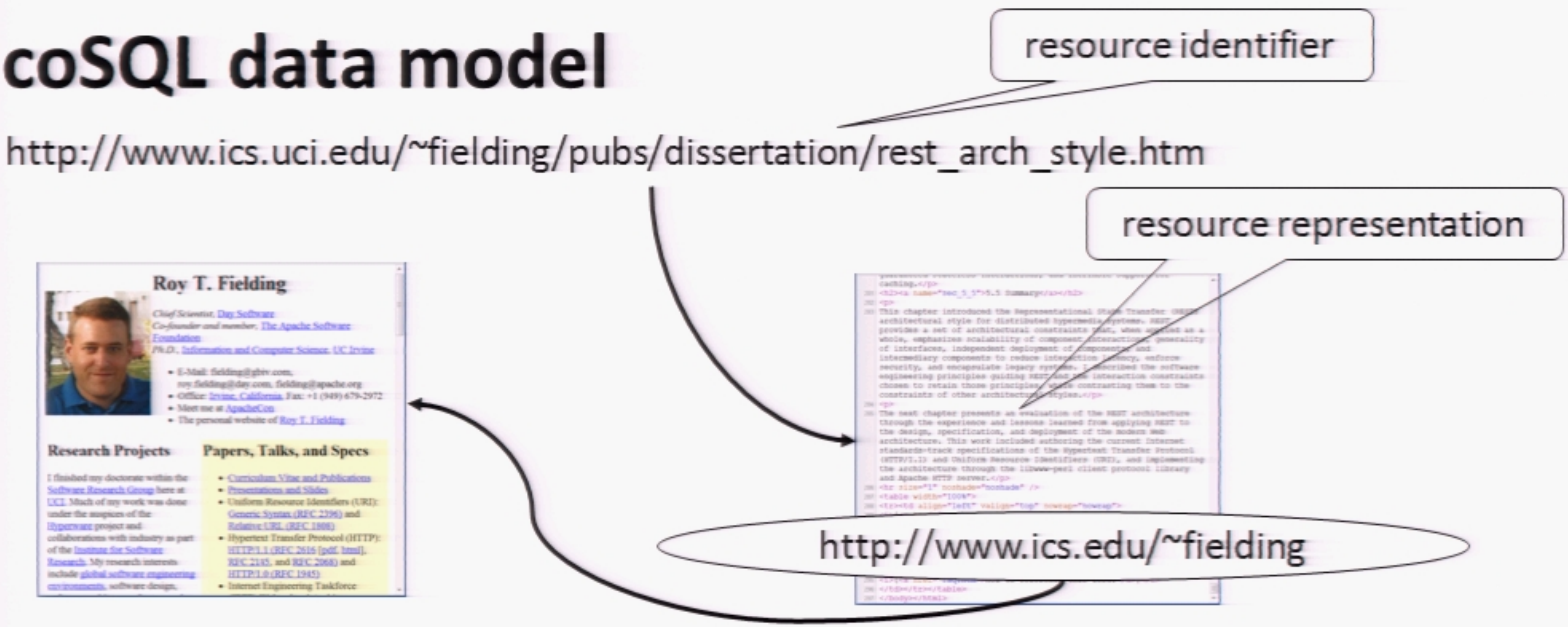
vs

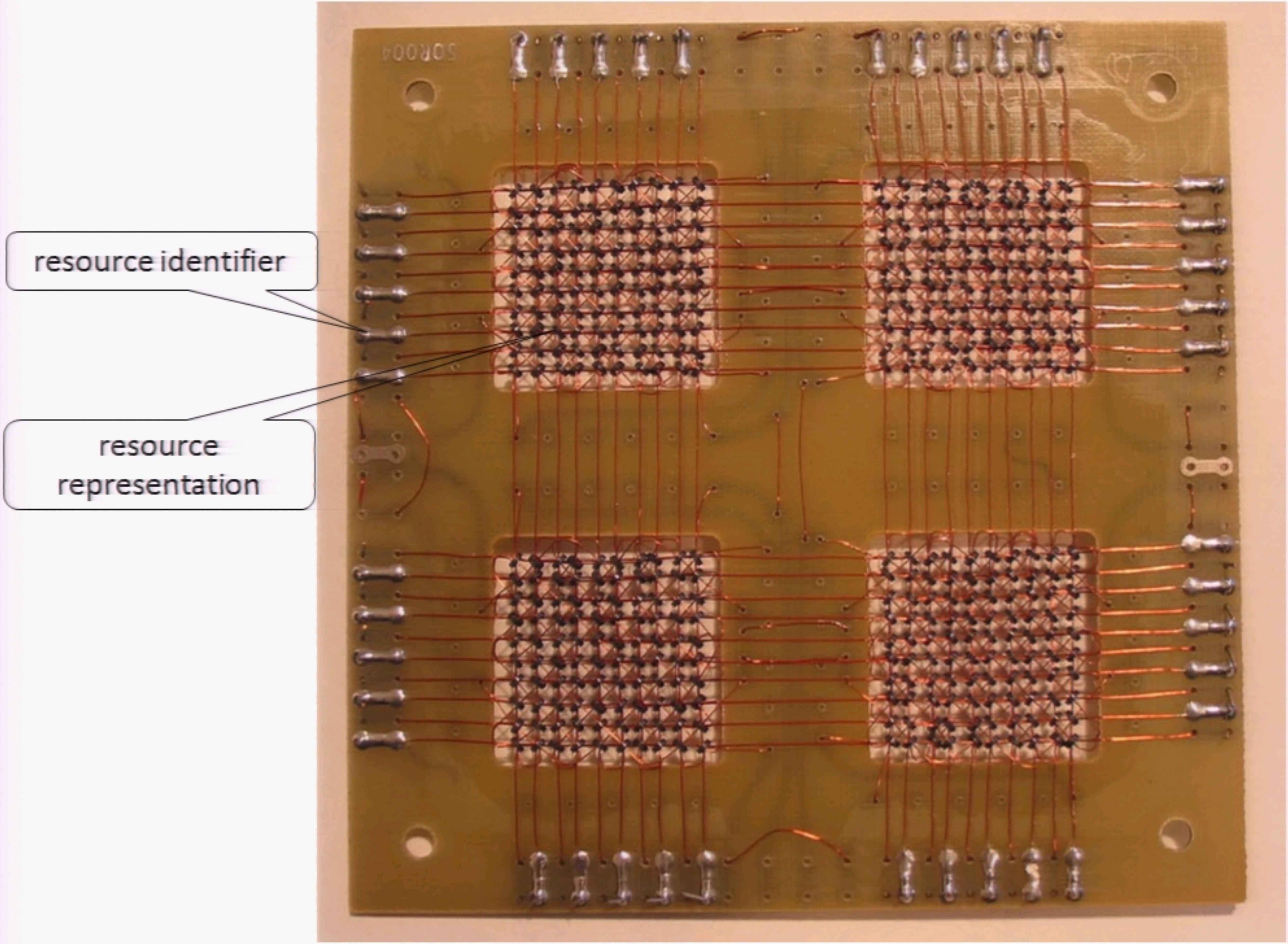
Children point to parent

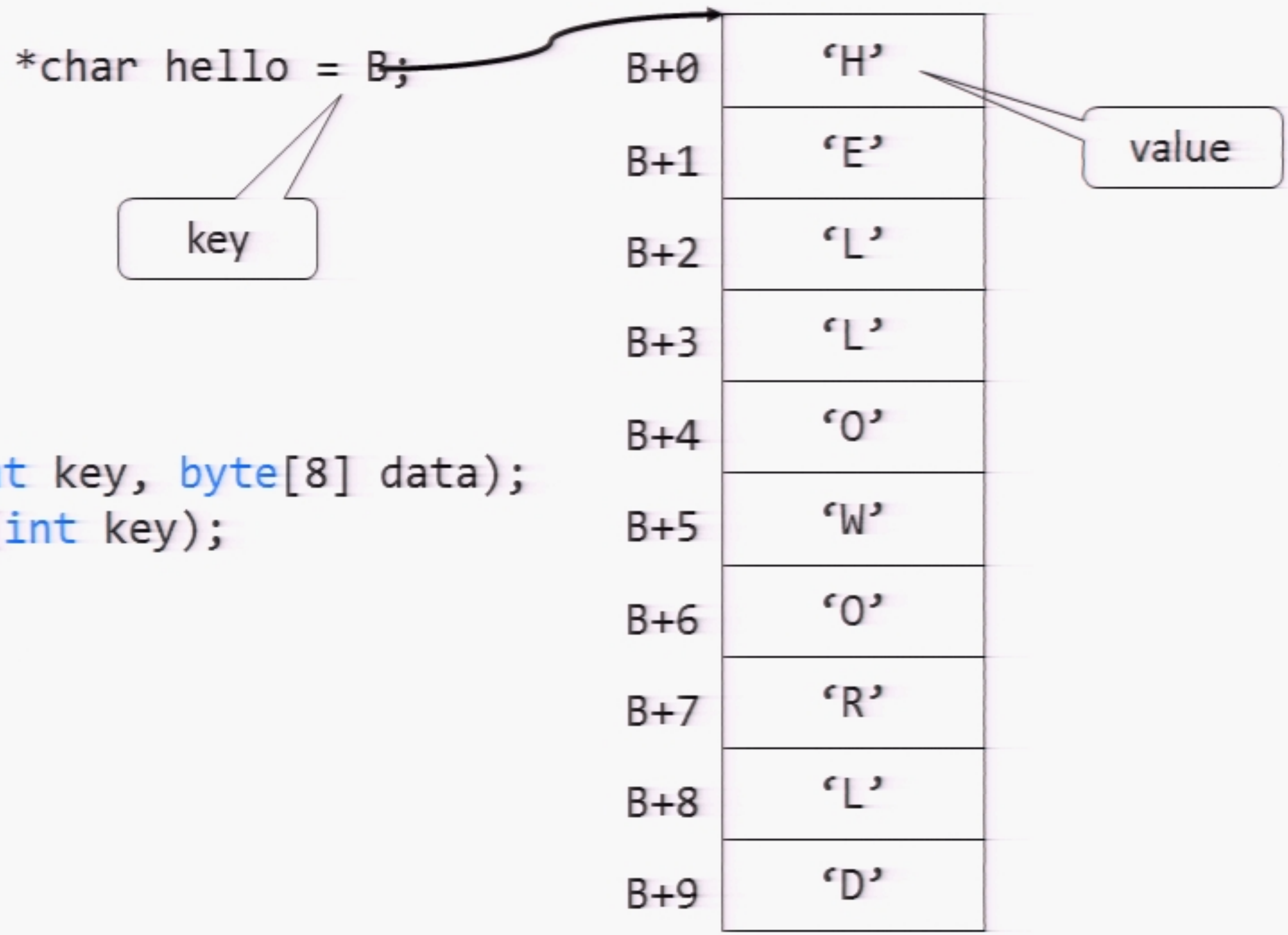
SQL data model



coSQL data model




resource identifier
resource representation




```
void Write(int key, byte[8] data);
byte[8] Read(int key);
```

Key-value store
more “natural” than
FK/PK store?

Neither is “universal”!

Consequence: typed vs untyped

Typed enough to determine FK and PK

```
void Write(int key, byte[8] data);
byte[8] Read(int key);
```

Both keys and values can be opaque

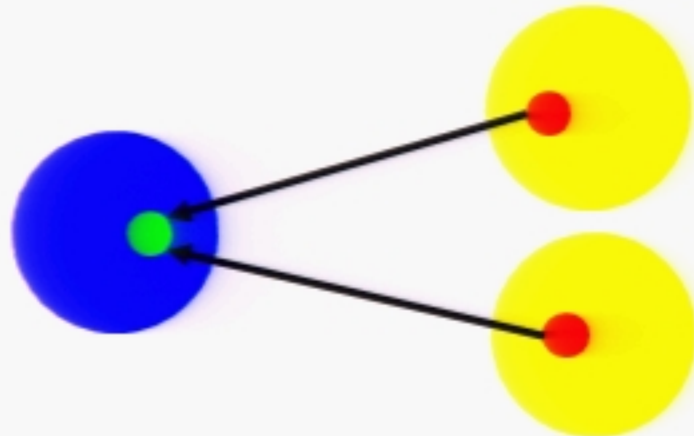
Consequence: value vs computation

To efficiently traverse association
and maintain referential integrity
must know values of PK and FK upfront

```
void Write(int key, byte[8] data);
byte[8] Read(int key);
```

Both Read and Write can involve
arbitrary computation, and potentially fail

Consequence: closed vs open

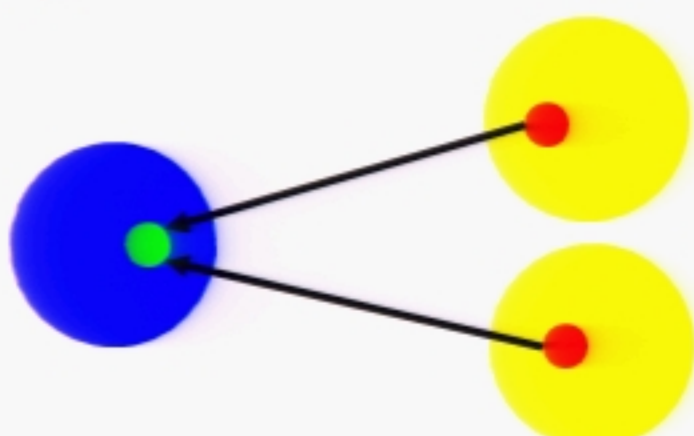


Must reason about all entities at once
(entity set, entity container, transactions, ...)

```
void Write(int key, byte[8] data);
byte[8] Read(int key);
```

Key space can be partitioned across many machines

Consequence: declarative vs imperative



Intimate knowledge of closed world
statistics-based query optimization
Highly available

```
void Write(int key, byte[8] data);
byte[8] Read(int key);
```

Decompose query in parallel Read/Write
Failure

Concrete Examples Of Collections/ Monads

Consumer *pulls* from producer

vs

Producer *pushes* to consumer

```
interface IEnumerable<out T>
{
  IEnumerator<T>
  & IDisposable  GetEnumerator()
}

interface IEnumerator<out T>
{
  bool MoveNext()
  T Current { get; } throws Exception
}
```

IEnumerable<T> =
() → IEnumerator<T>
& IDisposable

IEnumerator<T> =
() → T+()+Exception

Simplified: ()→(()→T)

$$\texttt{IObservable<T>} = \texttt{IObserver<T>} \longrightarrow () \;\&\; \texttt{IDisposable}$$

$$\texttt{IObserver<T>} = T \times () \times \texttt{Exception} \longrightarrow ()$$

Simplified: $(T \longrightarrow ()) \longrightarrow ()$

Continuation Monad!

```csharp
interface IObservable<out T>
{
    IDisposable
       Subscribe(IObserver<T> observer)
}

interface IObserver<in T>
{
  void OnCompleted()
  void OnNext(T value)
  void OnError(Exception error)
}
```

```
IObservable<T> Where(
  this IObserver<T> source, Func<T, bool> predicate)
{
   return Observable.Create<T>(observer =>
   {
      return source.Subscribe(Observer.Create<T>(
        value =>
        {
           try
           {
              if(predicate(value)) observer.OnNext(value);
           }
           catch (Exception e)
           {
              observer.OnError(e);
           }
        });
    };
}
```

CallCC

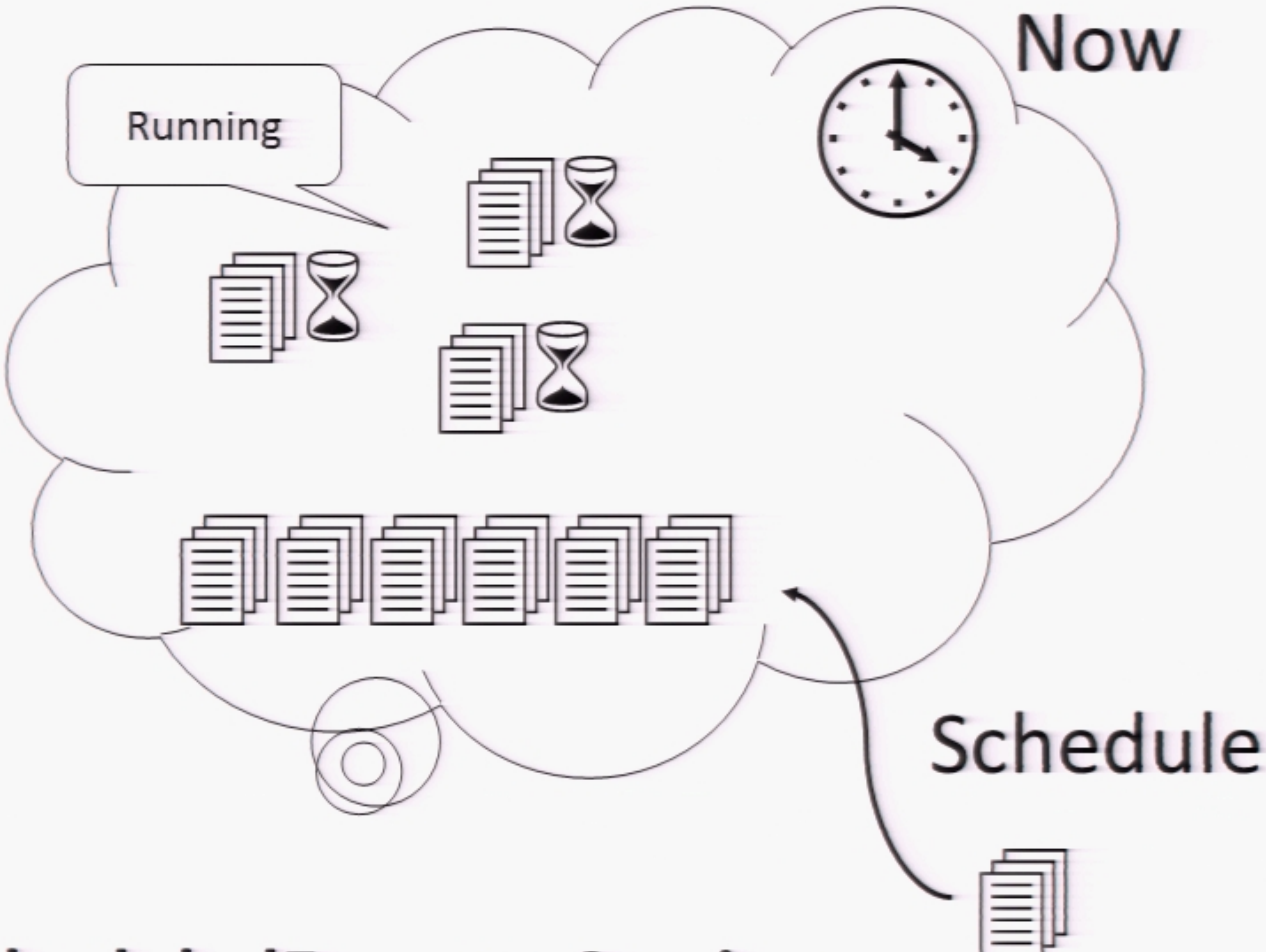
IEnumerable<T>

Introduce concurrency

$\cong$

Remove concurrency

IObservable<T>



Java ScheduledExecutorService

IScheduler

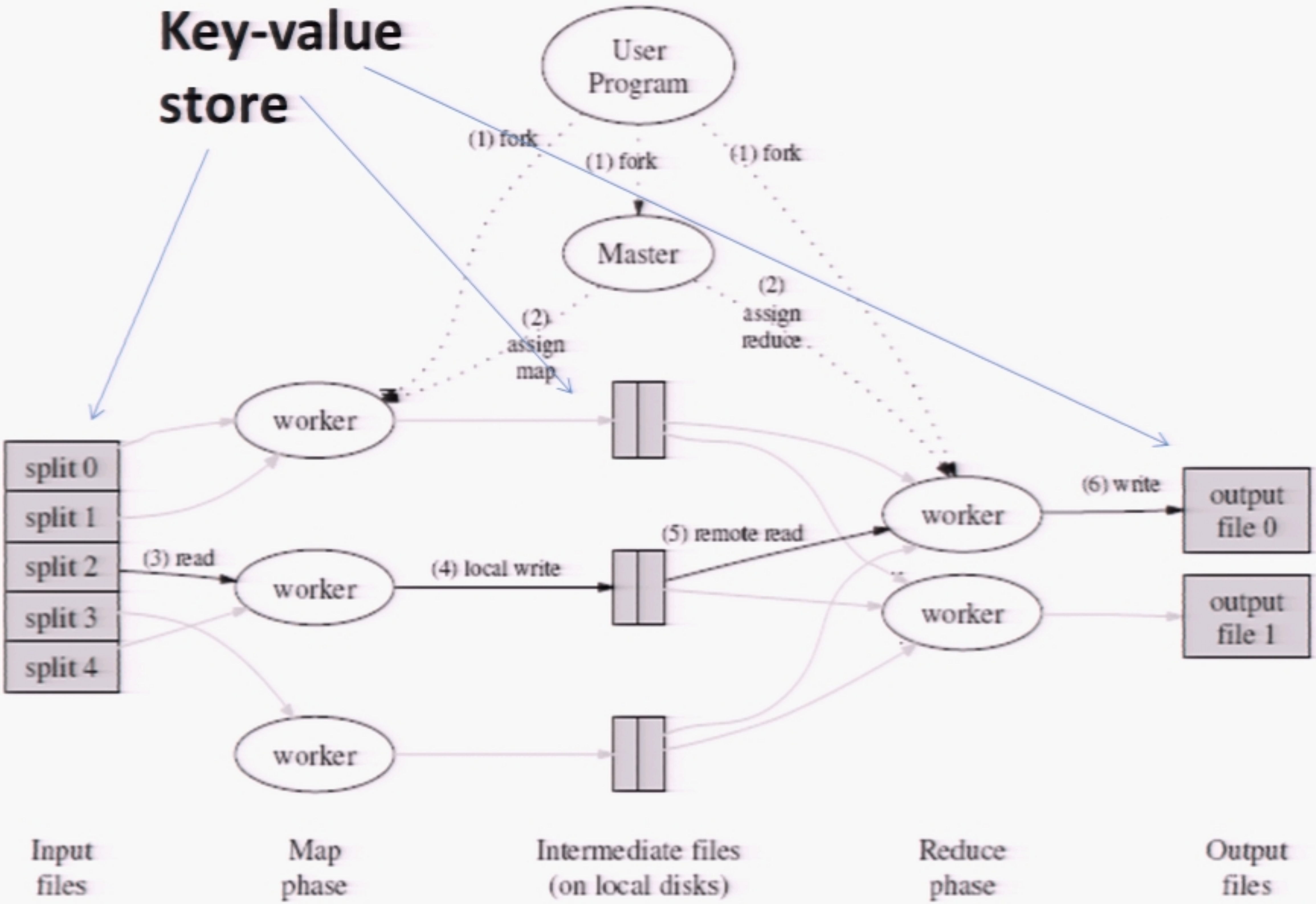
```
interface IScheduler
{
   IDisposable Schedule
    (Action work, DateTimeOffset when)

   DateTimeOffset Now { get; }
}
```

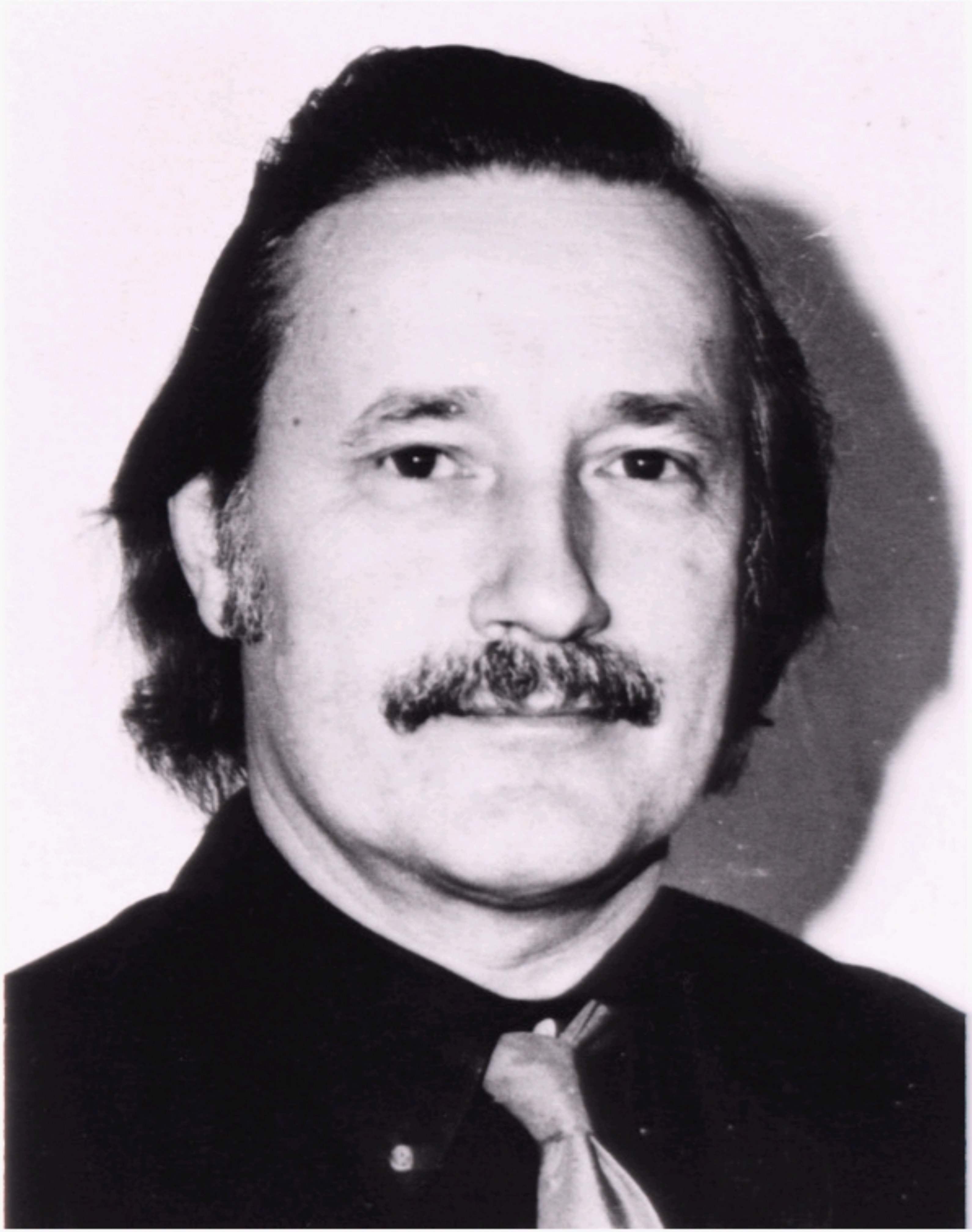
Simplified: $((\,)\overset{\text{action}}{\to}(\,))\longrightarrow((\,)\overset{\text{anti-action}}{\to}(\,))$

Circling back to the start

http://code.google.com/edu/parallel/mapreduce-tutorial.html



Compile LINQ queries into Map/Reduce Form



A Continuation Semantics for LINQ

Flatten nested queries into form ("left-deep join")

```
xs().SelectMany(
   x=>ys(x).SelectMany(
      y=>zs(x,y).SelectMany(
         z=>F(x,y,z)))))
```

```
Return(new{})
   .SelectMany(_   => xs(),          (_,x)  => new{x})
   .SelectMany(x   => ys(x.x),       (x,y)  => new{x.x, y})
   .SelectMany(xy  => zs(xy.x,xy.y),(xy,z) => new{xy.x, xy.y, z})
   .SelectMany(xyz => F(xyz.x, xyz.y, xyz.z))
```

Associativity for SelectMany

```
xs.SelectMany(x⇒(f(x).SelectMany(g)))
=
xs.SelectMany(x⇒f(x)).SelectMany(g)
```

```
xs.SelectMany(x⇒F(x).
                 SelectMany(y⇒G(x,y), K(x)), H)
=
xs.SelectMany(x⇒F(x), (x,y)⇒(x,y))
  .SelectMany((x,y)⇒G(x,y),
     ((x,y),z)⇒H(x,K(x)(y,z))
```

Abstract Syntax

Expr represents an expression of type X, Query represents an expression of type X^*

```
Expr ::= Constant
       | Param
       | Expr ⊕ Expr
       | Function(…, Expr, …)
       | Expr.Member
       | new{ …, Member = Expr, …}
       | Query -- box

Query ::= Source()
       | Return(Expr)
       | Query ⋓ Query
       | Query.Select(Param ⇒ Expr)
       | Query.SelectMany(Param ⇒ Query)
       | Query.SelectMany(Param ⇒ Query, (Param,Param) ⇒ Expr)
       | Expr -- unbox
```

`Expr ::= Query` ➔ query in expression context (“box”),
`Query ::= Expr` ➔ expression in query context (“unbox”).

Flatten a SelectMany query with free variables x

$\mathcal{Q}[\![\ ]\!] \in (X \to Y^*) \to (X \times Y \to Z) \to X^* \to Z^*$

$\mathcal{Q}[\![x \Rightarrow ys]\!]$ `f xs = xs.SelectMany(x⇒ys,f)`

Flatten a Select query with free variables x

$\mathcal{E}[\![\ ]\!] \in (X \to Y) \to (X \times Y \to Z) \to X^* \to Z^*$

$\mathcal{E}[\![x \Rightarrow y]\!]$ `f xs = xs.Select(x⇒y, f)`

Flatten a top-level query with free variables x

$\mathcal{M}[\![\ ]\!] \in (X \to Y^*) \to X \to Y^*$

$\mathcal{M}[\![x \Rightarrow ys]\!]$ `x = Return(x).SelectMany(x⇒ys, (x,y)⇒y)`

`= ` $\mathcal{Q}[\![x \Rightarrow ys]\!]$ `(x,y)⇒y Return(x)`

$\mathcal{M}[\![\ ]\!] \in (X \to Y) \to X \to Y^*$

$\mathcal{M}[\![x \Rightarrow y]\!]$ `x = Return(x).Select(x⇒y, (x,y)⇒y)`

`= ` $\mathcal{E}[\![x \Rightarrow y]\!]$ `(x,y)⇒y Return(x)`

$\mathcal{M}[\![\ ]\!] \in Y^* \to Y^*$

$\mathcal{M}[\![ys]\!]$ `= Return().SelectMany(()⇒ys, (_,x)⇒x)`

```
ℳ⟦x⇒ys⟧ = x⇒(𝒬⟦x⇒ys⟧ (x,y)⇒y Return(x))

𝒬⟦x⇒Return(e)⟧ f xs
=
  ℰ⟦x⇒e⟧ f xs

𝒬⟦x⇒ys.Select(y⇒e)⟧ f xs
=
 ℰ⟦(x,y)⇒e⟧ ((x,y),z)⇒f(x,z)
   (𝒬⟦(x,y)⇒zs⟧ (x,y)⇒(x,y) xs)

𝒬⟦x⇒ys.SelectMany(y⇒zs)⟧ f xs
=
  𝒬⟦(x,y)⇒zs⟧ ((x,y),z)⇒f(x,z)
     (𝒬⟦x⇒ys⟧ (x,y)⇒(x,y) xs)

𝒬⟦x⇒ys.SelectMany(y⇒zs, (y,z)⇒e)⟧ f xs
=
 ℰ⟦((x,y),z)⇒e⟧ f
 (𝒬⟦(x,y)⇒zs⟧ ((x,y),z)⇒ ((x,y),z))
   (𝒬⟦x⇒ys⟧ (x,y)⇒(x,y) xs))

𝒬⟦x⇒y⟧ f xs
=
  xs.SelectMany(ℳ⟦x⇒y⟧, f)
```

Derivation

```
𝒬⟦x⇒Return(e)⟧ f xs
=
xs.SelectMany(x⇒Return(e), f)
=
xs.SelectMany(x⇒Return(e).Select(y ⇒f(x,y)))
=
xs.SelectMany(x⇒Return(f(x,e)))
=
xs.Select(x⇒f(x,e))
=
ℰ⟦x⇒e⟧ f xs
```

Derivation

```
𝒬⟦x⇒ys.SelectMany(y⇒zs, (y,z)⇒r)⟧ f xs
=
(𝒬⟦x⇒ys⟧ (x,y)⇒(x,y) xs)).SelectMany((x,y)⇒zs, ((x,y),z)⇒r),f)
=
(𝒬⟦x⇒ys⟧ (x,y)⇒(x,y) xs)).SelectMany((x,y)⇒zs,((x,y),z)
⇒((x,y),z)).Select(((x,y),z)⇒r),f)
=
(𝒬⟦x⇒ys⟧ (x,y)⇒(x,y) xs)). (𝒬⟦(x,y)⇒zs⟧ ((x,y),z)⇒
((x,y),z)).Select(((x,y),z)⇒r),f)
=
ℰ⟦((x,y),z)⇒r⟧ f
   (𝒬⟦(x,y)⇒zs⟧ ((x,y),z)⇒ ((x,y),z))
      (𝒬⟦x⇒ys⟧ (x,y)⇒(x,y) xs))
```


I fint yur cat theory kool
Crock-Pot
i mai haz to uz it
ICANHASCHEEZBURGER.COM